# Instructions (ISAs)

## Table of content

- Definitions
- Computer Architecture
- CISC vs RISC
- Addressing Modes
- RISCV
    - Instruction Format
    - Basic Architecture
    - Types of Instructions
    - Data Transfer Instructions
- External Links

## Definitions

- ISA (Instruction Set Architecture)
    - Dependent on hardware. Varies with system
    - RISC (Reduced Instruction Set Computer) ex - ARM
    - CISC (Complex Instruction Set Computer) ex - Intel, AMD
- Addressing mode
    - Indirect addressing - when pointer is used to access the memory
    - Direct addressing - Direct values are given
- Blocks
    - ALU - The arithmetic block used to perform operations. They are fed direct values after resolving the pointers
- Performance
    - Throughput - inversely proportional to the clock delay for an instruction. This value decrease as the complexity of each increases
- Data types
    - double word - 64 bits (8 bytes)
    - word - 32 bits (4 bytes)
    - half word - 16 bits (2 bytes)
    - byte - 8 bits
- Instructions
    - `lw` - load word, is used to read memory from the data memory. It functions by adding an offset value to the address that is given as a parameter

        - ex. `lw rd, imm(rs1)`
            - Here, we offset the address in rs1 by `imm` value and access the memory from that address.
            - So `lw rd, 2(rs1)` adds a memory offset of 2 to the address stored in rs1 and stores the value from the that location in the memory into the `rd` register.
    - `sw` - store word, is used to store a value into the data memory location. It functions similar to `lw` except for storing into the memory instead of reading
        - ex. `sw rs2, imm12(rs1)` or `sw x5, 0x04(x10)`
- Types memory structure
    - Byte addressable
        - Addressed in terms of a single byte
        - This format is used to support more data types
    - Word addressable/ double word address
        - Addressed in terms of a word (32 bits / 4 bytes)
        - Not used often because it's inefficient for data types of smaller sizes

## Computer Architecture

- Types of classification
    - Order of storage
        - Little Endian
            - Least significant bit stored first
        - Big Endian
            - Most significant bit stored first
    - Data Path
        - Harvard
        - Von Neumann
    - Complexity
        - RISC
        - CISC

## CISC vs RISC

| Features | CISC | RISC |
|---|---|---|
| Memory & ops | Direct memory access- The result from an operation is send back to the memory block for storing using pointer | Load - store - The results are stored in registers |

| Features | CISC | RISC |
|---|---|---|
| Instruction Length | Lesser Instruction Length | More instruction length |
| Complexity of instructions | More complex instructions. (Capable of resolving pointers automatically. Hence lesser code length) | More simpler instructions. Uses register for it's operation. (ex. Hence requires more code to resolve pointers) |
| Addressing mode | More types | Lesser types |
| No. CLK cycles for execution of instructions | Variable throughput (Based on the complexity of instructions) | Fixed throughput (Registers make the throughput uniform) |
| Encoding | Variable instruction encoding size because of different addressing modes | All instructions are encoding using 32 bits |
| Pipelining | | |

## Addressing Modes

- Immediate
  - These are hard coded constants in the code itself. They are not stored in a specific register. Instead they are directly send to the execution unit (ex. ALU).
  - ex. `add x5, x6, 0x10`
- Direct
  - These are variables in an assembly code. These values are stored in specific registers before processing/execution
  - ex. `add x5, x6, x7`
- Indirect
  - These use pointer like method for accessing data in the memory. The address is used to point to the memory location of the actual data. This location need to be stored in a register to be called indirect addressing.
  - ex. `add x5, x6, [x7]`
- Complex
  - Both indirect and direct can be used together to represent the physical address of the memory
  - ex. `add x5, x6, [x7 + 0x10]`

**Note**

*Physical Address*

- PA (Physical address) = BA (Base address) + offset

- ex. This is used in the program counter to set the next address which contains the next instruction. The offset is determined by the size of each instruction.

**Note**

*More clarification on direct vs indirect*

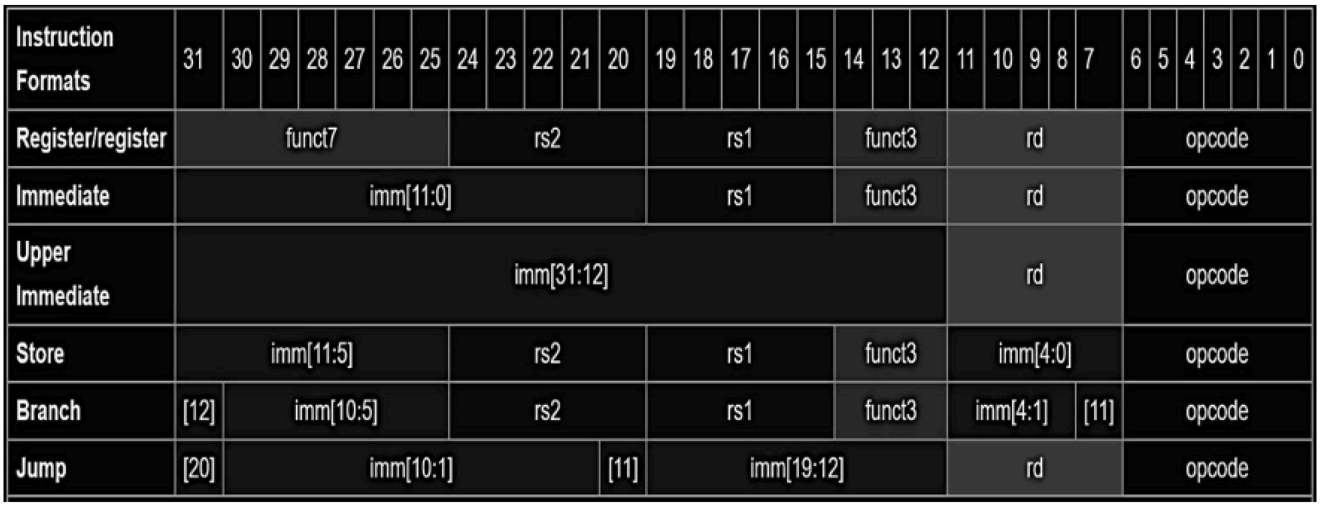
- `[x2]` is direct addressing as it need to extract the physical memory location from a register and decipher the physical address before the data can be accessed.
- `[1000H]` is not indirect addressing as it doesn't require an extra step to retrieve the physical memory address. This would come under direct addressing

# RISCV

## Stages

- Instruction fetch
  - Machine code is fetched from memory location pointed by PC
  - `PCnew = PCold + 4`
- Instruction decode
  - R-type: Contents of 2 source registers (as Reg 1 and Reg 2) are read and given to next stage
  - Load/Store /I type instruction: base register is read as Reg 1 and 12 bit immediate value as offset
- Execution
  - R-type: Arithmetic / Logical operation on `opr1` & `opr2`. `ALUout` register stores result
  - Load/ Store: the address of data memory is generated by adding base register with Imm value and then stored in `ALUout` register
- Memory access
  - LOAD: `ALUout` is address of Data memory location, and content is read
  - STORE: `ALUout` is address of Data memory location, and content of source register is sent to the memory location
- Write back
  - R-type: content of `ALUout` is stored into the destination register
  - Load: data read from memory is stored in destination register

## Instruction Format



| Instruction Formats | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Register/register | funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | |
| Immediate | imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | |
| Upper Immediate | imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | opcode | | | | | | |
| Store | imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:0] | | | | | opcode | | | | | | |
| Branch | [12] | imm[10:5] | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1] | | | | [11] | opcode | | | | | | |
| Jump | [20] | imm[10:1] | | | | | | | | | | [11] | imm[19:12] | | | | | | | | rd | | | | | opcode | | | | | | |

- Fields
  - Opcode (7 bit): Determines the operation to be perform based on the instructions given in the code
  - fun3 (3 bits), fun7 (7 bits): depicts the type and operation to be performed ex: SUB/ ADD with same adder
  - destination register:
    - register
    - data memory
  - source fields:
    - register
    - data memory
    - immediate value/ constant
- Size
  - CISC - The binary encoded instruction size is variable based on the instructions and addressing types. This is cause each instruction is complex and it would be inefficient to use same instruction size.
  - RISC - The binary encoded instruction size is kept constant irrespective of the instructions. This is possible because the instructions on RISCV is simple (perform simple operations) and it supports limited number of addressing modes. (ex. RV32 uses 32 bits for each instruction)

**Question**

- What will be the number of extra bits required to encode 128 registers instead of 32 registers in R-type instructions of RISCV ?
  - 38
  - [✓ ~~6~~]
  - 8
  - 32
- What will be the encoding bits (number of instruction) bits required if the number of register are increase similar to previous question ?

[✓ ~~38~~]

○ 6

○ 8

○ 32

## Basic Architecture

### Memory

- Code memory
  - Holds all the instructions from the code in binary format
- Data memory
  - This memory holds all the data used for the execution of the instructions

### Program Counter

- Register that is used to determine the execution flow of the processor. This register holds the address of the code memory location in which the next instruction to be executed is stored.
- It functions on the basis of offsets. So in a linear execution flow (i.e. without if statements and loops) after every execution of an instruction the register is updated to the next address by adding an offset (equal to the instruction bit size) to the current address. (This is in RISC architecture cos of the constant instruction size)
- On reset PC always holds address of the 1st instruction of program.

### Datapath

- Collection of state elements, computation elements, and interconnections that together provide a part for flow and transformation of data in processor during execution.

### Registers

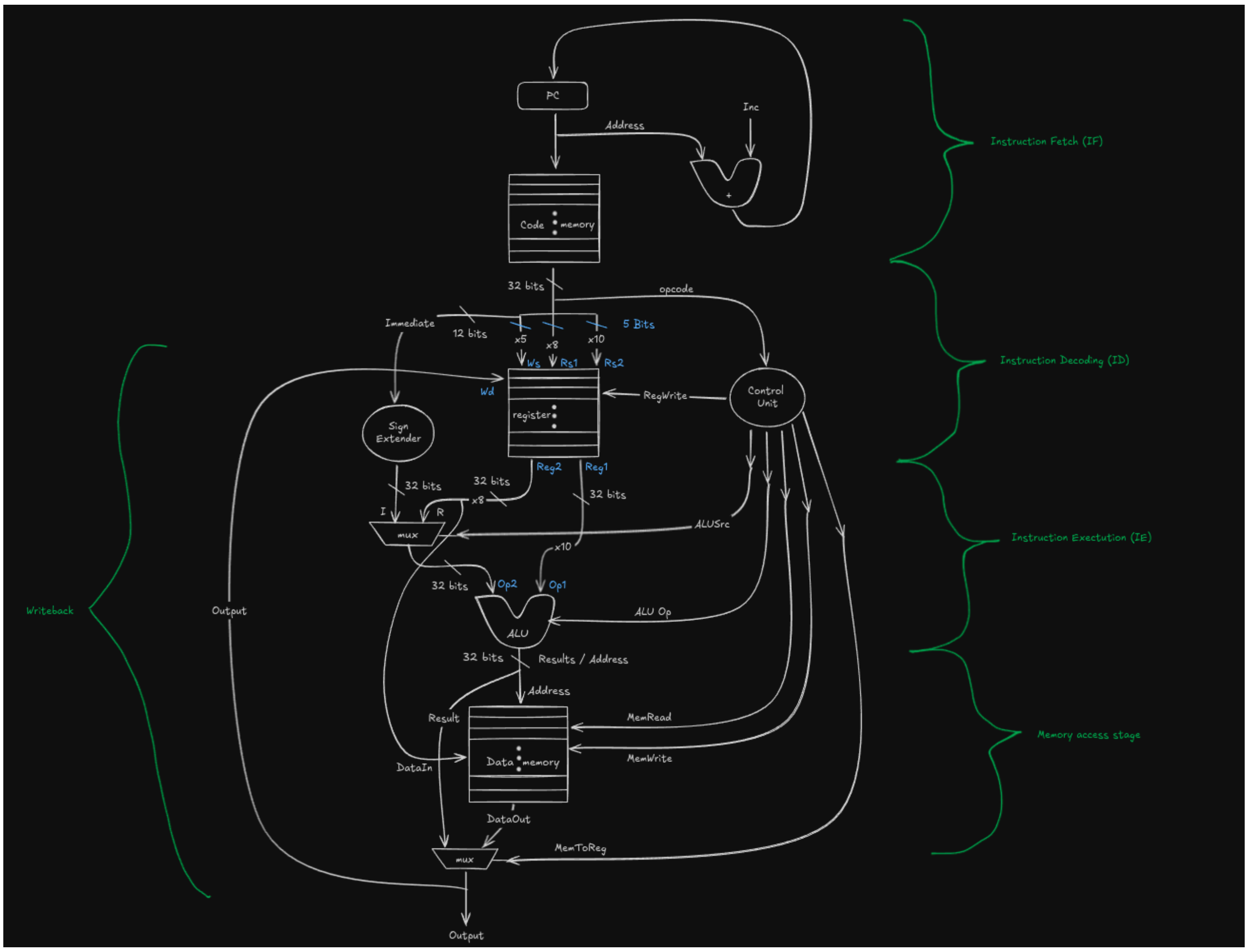
**Note**

All register names are named with x[number]
ex. x10 <=> 10th register

- rs - source register

- rd - destination register
  - ex. `add rd, rs1, rs2`
- Types
  - Zero register
    - Shorted to ground, user doesn't have any control of the this register. It will always store the value of 0.
  - Save registers (`s0` - `s11`)
    - Used to hold variables
  - Temporary register (`t0` - `t6`)
    - Used to hold intermediate values during a larger computation
  - Function arguments / return values
  - Pointers
    - Used to point to a specific code memory.
    - For example after executing a sub-process, the global pointer stores the instruction the control has to come back to after computation
    - ex. Stack, Global, Thread

## Structure



**Note**

WD - Data bus (Destination)
WS - Address bus (Source)

## Types of Instructions #todo

- R-Type (Register)
  - In this type of instructions, only data from the registers are used for execution of instructions.
  - Instructions
    - `add`
    - `sub`
    - `and`
    - `or`
    - `xor`
- I-Type (Immediate)
  - In this type of instructions, data from the registers and immediate value from the code are send to the ALU for computation.
  - `imm` value can be any 12 bit number.
  - Instruction
    - `addi`
    - `andi`
    - `xori`
    - `ori`
  - L-type (Load)
    #todo : What type of instruction it come under?
    - ex. `lw rd, imm12(rs1)` or `lw x5, 0x04(x10)`
    - PC -> Code memory -> CU, Register, mux -> ALU -> MemRead=1, mux -> Result -> WD (Register)
    - 5 Stages
    - Instructions
      - `lb`
      - `lh`
      - `lw`
      - `lbu`
      - `lhu`
- U-type (Unsigned)
- S-type (Store)
  - ex. `sw rs2, imm12(rs1)` or `sw x5, 0x04(x10)`

    - Instructions
        - `sb`
        - `sd`
        - `sw`
- B-type (Branch)
- J- type (Jump)
  - Instructions
  - `j label`
  - `jal rd, label` - jump and link - range $2^{20}$
  - `jalr rd, imm(rs1)` - jump and link register

> **Question**
> how does branch not require a return register

- Example Code
    - R-type

    ```
    lw x5, 0(x10)
    lw x6, 4(x10)
    add x7, x5, x6
    ```

    - I-type

    ```
    lw rs1, (x10)
    addi rd rs1, imm12
    lw rd, imm12(rs1)
    ```

> **Note**
>
> In an ALU the operand 1 (Op1) is always from the register and operand 2 (Op2) can be from a register or immediate values depending on I or R types instruction.

> **Note**
>
> When it come to ops like add IMM and R, the memory access stage is kept idle.

> **Summary**
>
> *I-type*

- We can call an instruction I-type is it has operands of rd, rs1 and immediate 12-bit number.
- The `sw` instruction is an I-type instruction because we use an offset along with the base address in the register to store the data

*R-type*

- We call an instruction R-type if it's operands are all registers and there is no immediate value involved
- Arithmetic instructions are R-type instructions because the data is always taken from registers only to be operated on.

## Data Types

- byte, half word, word, double word
- unsigned no.s/ signed no.s

## Design Principles

- Simplicity favours regularity
  - It's easier to replicate the same operation multiple time than to create a completely create a new one from scratch
- Smaller is faster
  - More number of register increases the clock cycle time because it takes electric signals longer to travel
  - The number of bits it would take in instruction format to address a register.

## Types of RV

- RV32G
  - A combined set of IMAFD extensions
    - I - Integer
    - M - Multiplication
    - A - Atomic
    - F - Float (single precision)
    - D - Float (Double precision)
  - 32 bit representation
- Compressed #todo
  - 16 bit representations
  - 2 byte instructions extension

**Note**

RV32 and RV64 has 32 bit instruction size. The difference is the size of ALU and more instructions to deal with double words. like the `ldw`

## Encoding

**Question**

Make the following into binary encodings

```
add rd, rs1, rs2
rd rs1, imm(rs2)
sw rs2, imm(rs1)
```

| fun7 | rs2 | rs1 | fun3 | rd | opcode | Hex |
|---|---|---|---|---|---|---|
| 0000_000 | 0_1010 | 1001_1 | 000 | 1001_0 | 011_0011 | 0x00A98933 |
| | | | | | | |

## Sign extension of immediate 12 bit numbers

- Immediate instructions in RISCV uses 12 bit signed immediate values. Therefore during such operations, the 12 bit number are sign extended
- For example
  - 0x744 -> 0x0000_0744
  - 0x801 -> 0xFFFF_F801

  **Question**
  What is the result stored in the destination register ?

```
addi x15, x12, 0x7FF
addi x5, x12, 2049

addi x5, x0, 0x744
and x5, x0, 2049
```

## Shifting Operators

- Right shift ( $/2$ )

    - Logical - doesn't maintain signs of number
    - Arithmetic - maintains the sign of the number
- Left shift ( $\times 2$ )
  - Logical - Shift left

> **Question**
> Divide -88 by 8 using only shift operators

```
.data
a: -88

.text
la x10, a

# it can also be stored like
# addi x10, x0, -88
srli x11, x10, 3 # logic shift, the sign of the number is not maintained
srai x11, x10, 3 # arithmetic shift, the sign of the number is maintained
```

> **Question**
>
> Convert C code into asm. `B[8] = a[i-j]`
> Assume the variables
> f -> x5
> g -> x6
> h -> x7
> i -> x28
> j -> x29
> Base Address of a -> x10
> Base Address of b -> x11

```
sub x30, x28, x29
slli x30, x30, 2
add x31, x30, x10
lw x30, 0(x31)
```

> **Note**
>
> If the indexes are know use following steps to calculate

        - shift the index register towards left by depending on size of word (Byte, 1 - Half, 2 - word, 3 - double)

## Branching #todo

- If statements with goto
- Needs a sign extended 13 bit number (unlike the normal 12 bits) to increase the range of branching

## Support for extended immediate values #todo

- 2 rules
    - Divide the constant into Imm 12:31 and Imm 0:12
    - Is the constant give signed or unsigned
        - If Unsigned -
        - If Signed -
- Addressing modes define different ways to define 2 operands
- Addressing on branches
    - PC relative
- Instructions
    - `lui`
    - `auipc rd, imm[31:12]`
        - used to calculate the base address of symbol defined in data segment using PC relative addressing mode
        - Used for calculating the long target address of symbol beyond the range of jal

## Procedures in Computer Hardware

- Put the parameters to pass to the subroutine(procedure) into the argument register (a0 - a7)
- After a subroutine (procedure), the register values which were backed up into the stack memory is retrieved creating the concept of localisation of variables in a function

## Stack

- SP - Stack Pointer used to indicate the allocated memory used by the stack
- LIFO architecture

- No dedicated instructions for pushing and popping in a stack. We use the same `lw` and `sw` instructions

## External links

- Excalidraw

> 1. To open excalidraw files, download the the `.excalidraw` file and head to the site
> 2. Click on the menu icon in the top left corner and open the file.

- ProjectF RISCV cheatsheet
- RISCV Extension List Gist
- RISCV Instruction Groupings
- COD Codes

## Questions

- assume the base address of variable k in the data memory to be accesed is 0x8000_0020. PC present -> 0x0000_000c. Write the program to initialise the base address in x11, without using psuedo instructions
- Assume the address of symbol is 0x7FFF_F018 PCpresent is 0x0000_0008. WAP to change the control of execution to address of symbol

1. $$relative\ address\ =\ base\ address\ -\ PCpresent$$
$$relative\ address\ =\ 0x7fff$$